Between the Lines

Bar Code and Decoding Bar-code Algorithms

No matter where you look, it seems bar codes are becoming more and more prevalent. John takes a look at a few of the many different encoding schemes in use and their history.

EMBEDDED TECHNIQUES

John Dybowski

Automatic Identification or Auto ID, for short, revolves around the art of decoding various machine-readable symbologies. This art is far faster and more accurate than manual data entry and therefore serves as a superior alternative. Although there are several machine-readable media, the most popular and enduring is bar code. Its ease of production, proven reliability, and especially its low cost make it a prime solution.

A bar code can be viewed as a form of paper memory, a printable machine language, or a machine-readable document. Any way you look at it, bar code easily translates directly into bitstreams of ones and zeros, the substance of modern computers.

With greater amounts of micro-electronic integration, miniature computers are everywhere, often masquerading as appliances. Bar-code readers, having attained appliance status, can be placed anywhere data capture is convenient or necessary. Combined with hand-held or stationary, contact or no-contact bar-code scanners, they function as front ends for automated data-collection systems.

In extreme cases, small computers are embedded in hand-held bar-code wands providing intelligent, stand-alone readers capable of outputting RS-232 datastreams. These datastreams are suitable for direct input into larger, data-processing computers.

Although a survey of the 50 or so prevailing bar codes reveals that some relatively weak encoding methods have survived, their numbers are few. They exist for historical reasons. That is, because of their rapid early adoption and large installation base, they became entrenched in specific application areas.

As a general rule, however, the industrial and financial sectors are firmly based on pragmatic footing. Poor performers just don't last.

Let's start with an overview of some alternate data-capture methodologies so we can get a better appreciation of bar code's simplicity.

KEYBOARD ENTRY

By definition, Auto ID is keyless data entry. Except as an auxiliary input device, the keyboard is gone. It is only used to enter supplementary information and as a backup if the encoded symbology cannot be read.

OCR—NICE TRY

Optical character recognition (OCR) applies electrooptical techniques to machine-read printed characters which are also humanly readable. OCR is attractive because the same printed characters can be read by people and machines. Using a matrix of phototransducers, the illuminated symbol is scanned, usually with a hand-held device. As the scan head travels across the label, the presence or absence of reflected light indicates the presence or absence of portions of a character. Once the scan is completed, the acquired optical characteristics are evaluated, and (hopefully) the scanned character is identified.

Unfortunately, the key word is hopefully. If you've watched someone labor at scanning an OCR label, then you know what I mean. The problem, of course, is that the information available to decode OCR is very sparse. If a void, spot, or smudge is present, information is often misread or not decoded. As well, OCR has a problem distinguishing similar characters.

To counter this ambiguity, the National Retail Merchants Association (NRMA) has developed two OCR standards. In OCR-A, machine readability is maximized by making each character as different as possible from all others. Although this helps with the problem of machine-induced substitution errors, it follows that the unfamiliar appearance of the OCR-A alphabet results in an increase in human-induced errors.

To make the character set more humanly pleasing, OCR-B was developed. Unfortunately, OCR-B cannot be machine read with the same assurance as OCR-A. Since it is impossible economically to maximize both human and machine readability using the same character set, machine and human readability are, by nature, contradictory.

Font	Character Set	Application
OCR-A Numeric subset (A)	0123456789 |¥Ƒ∂-	Font distorted to maximize machine readability
OCR-A Numeric subset (B)	0123456789 |$.+-	
OCR-A Alphanumeric	0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ {}%ƒ&"*÷$,.-/'=|;:■	
OCR-A Numeric	0123456789+,-./|■	
OCR-A Alphabetic	ABCDEFGHIJKLMNOPQRSTUVWXYZ {}%ƒ&"*÷'=|■:;	
Farrington 7B	0123456789	Used on credit cards
E13B (MICR)	0123456789⑆⑇⑈⑉	Used by banks on checks
OCR-B Numeric	0123456789 < >+-/|	Enhanced appearance to human eye
407	0123456789|+$.-/ ¤	
1428	0123456789|,/-+H	

Figure 1—*OCR characters are used when it's necessary that both humans and machines be able to read the coded message.*

Why has so much effort been expended refining a technology that cannot deliver? Technological improvements can yield incremental OCR performance gains, but it's primarily based on economics, not technology. If you consider the inherent problems and the necessary tradeoffs, the whole concept looks like it's based on shaky ground. For your analysis, several OCR fonts are depicted in Figure 1.

MAGNETIC STRIPES

Recording data on a magnetic stripe results in a higher bit density that can easily be produced using a printing process on paper. Magnetics also offers the capability of altering or rewriting the recorded data after it's laid down. Although this rewriting capability is useful in a number of applications, it is actually a disadvantage in applications where data security is important.

Most commonly, magnetic stripes are read by passing an encoded card manually through a slot reader or by using a mechanized, insertion-reading device. Alternatively, a hand-held wand reader can be passed over a stationary magnetic stripe.

The primary disadvantage of magnetic media is its inability to be inexpensively printed. Advantages include a very high bit density and, for certain applications, its read/write capability.

TOUCH MEMORY

Touch memory, a fairly recent addition to the Auto-ID field, comes in a variety of forms. Available in read-only and read/write configurations, touch technology can serve applications that otherwise use OCR, magnetic stripe, or bar code.

The storage medium is silicon rather than ink-on-paper or the magnetic-flux reversals of the technologies I've already described. Silicon offers some intriguing possibilities. The fundamental device is ROM based and functions as a silicon number tag.

Other devices add nonvolatile RAM to the basic ROM configuration for storing changeable data. This provides numerous variations for high- and low-security storage regions. As well, they can be equipped with a real-time clock, a feature that opens applications which are otherwise simply unattainable.

A touch reader is easily the least expensive of the data-capturing technologies. It would be a great deal if your system had a lot of readers and just a few touch devices. However, this is seldom the case. Unfortunately, savings at the reader end are offset by the relatively high cost of the touch devices themselves.

The touch microcontroller interface consists of a single bidirectional port pin and a mechanical probe. The probe contacts the touch device's case, which resembles a small coin cell. The outer shell is the return connection and the center contact is the data connection. This simple and rugged case design is one of the touch device's principal features since this hermetically sealed, stainless-steel case survives hostile environments that would quickly turn paper or magnetic tags to pulp.

BAR CODE

Bar-code symbology provides the right feature mix for a lot of data-capture tasks, including identification of objects, locations, and people. The first step in deciphering a bar code involves acquiring the optical bar or space pattern using some form of electrooptical scanning device.

Let me briefly touch on some of the more common input devices before moving on.

Bar-code scanners can be hand held or stationary, fully automated or require a human operator, and they can operate in the visible or infrared spectrum. The least expensive way to scan a bar-code label is with a hand-held wand or a light pen. This device contains a built-in light source and a sensor for detecting the light reflected from the bar code.

Modern bar-code wands translate the optical symbol into a digital representation. To do this, they have the required amplification and wave-shaping circuitry built in. Optical slot readers operate essentially on the same

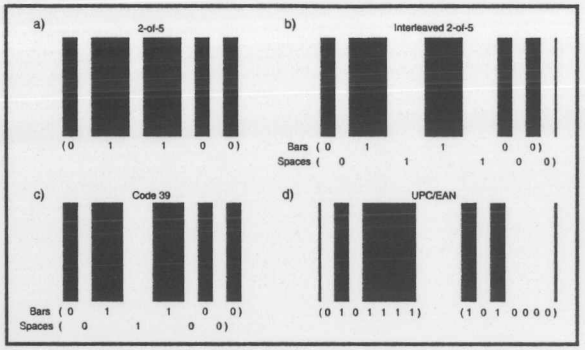

Figure 2—*These four common bit-encoding methodologies all depict the character 6. UPC goes one step further and defines both left- and right-handed symbols for the same character (d).*

principal, but require bar code to be passed through a slot in a manner similar to a magnetic card.

Fixed-beam scanners require bar code to be moved past the scanner. These can range from very small devices designed to read high-density codes to large units that read containers as they pass along a conveyor belt.

Moving-beam scanners can be hand-held or stationary. A number of light sources have been used, but modern devices almost exclusively use laser light. Helium-neon or solid-state lasers are most often used. A great depth of field, combined with a rapid, repetitive light sweep, results in a much higher "first-read rate" than a single-pass scanner offers. This improved first-read rate is also the result of reading the symbol continuously until it comes out right.

Scanners based on charge-coupled device arrays (CCD) replicate the function of a moving-beam scanner without moving parts. Using such a linear array, a solid-state scan is achieved by flooding the symbol with light and reading the transducer-array outputs sequentially. The resulting datastream is accomplished without physical contact or relative motion between the symbol and the read head.

ENCODING TECHNIQUES

Bar codes are composed of a series of dark and light bars that represent letters, number, and other symbols. These dark and light bars (or bars and spaces) are organized according to the specific rules of a particular bar-code symbology. In contrast to OCR, bar codes are conceived as a machine language designed for computers.

Bar code uses the ones and zeros fundamental to digital logic and is essentially binary in form. That's not to say that human-readable characters may not be contained on a bar-code label. But, the characters are provided for information purposes only; there is no attempt to electrically decipher them. The computer and human information is kept segregated—a lesson learned from OCR. Similarly, for increased efficiency, bar-code symbologies are optimized for a specific application. Tradeoffs are made between conflicting properties.

I'll be looking at several of the more popular and useful, not to mention simpler, bar-code structures a little more closely. But first, let me cover some fundamental concepts.

The smallest element of a bar code is a *module* (also sometimes referred to as the *X dimension*). In most cases, the wider bars and spaces are integer multiples of a module. Clearly, these relationships remain consistent as the codes are magnified or reduced in overall size and as they are scanned at different velocities.

Modules translate optical bar code into a binary code. Ones and zeros are extrapolated from the bar-space pattern which varies according to the specific bar code. In some cases, wide

CIRCUIT CELLAR INK
THE COMPUTER APPLICATIONS JOURNAL

READER SERVICE
Dec. 1994 effective through Feb. 28, 1995

Name _____
Address _____
City _____
State _____ Zip Code _____
Company Name _____ Phone # _____

Advertiser's Service

101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240

IRS—INK Rating Service

400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419
420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439
440	441	442	443	444	445	446	447	448	449	450									

New Products Service

500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519

INDEX

	RS#
Parallax, Inc.	•
Payne Research Inc.	166
Photronics Research, Inc.	106
Prairie Digital, Inc.	162
ProBoard Circuits	•
PseudoCorp	188
Pure Unobtainium	•
RAVEN Computer Systems	•
RC Systems, Incorporated	•
Real Time Devices	118
R.E. Smith	164
Rigel Corporation	130
Senix Corporation	192
Softools, Inc.	177
Software Science	165
Suncoast Technologies	155
Systems Engineering Associates, Inc.	190
Systronix	112
Technological Artisans, Inc.	184
Technological Arts	176
TechTools	193

Character	Code
0	00110
1	10001
2	01001
3	11000
4	00101
5	10100
6	01100
7	00011
8	10010
9	01010
Start	110
Stop	101

Table 1—*Two-of-Five encoding defines just the numbers 0–9 and unique start and stop symbols. The code has been widely used in industrial applications since the late '60s.*

elements translate to ones and narrow elements, to zeros.

Other bar codes assign binary values to dark and light elements. In such a scheme, a dark bar, which spans several module dimensions, accumulates the respective number of one bits. Similarly, zeros are accrued using spaces of varying width.

There are different ways these fundamental bit-encoding techniques are applied to creating a bar-code character set. Some bar codes use only the bars to represent data bits, with the spaces functioning merely as separators. Others use both bars and spaces to form a single-code representation of a character. Another method using both bars and spaces interleaves the coded characters so that bars encode odd characters and spaces represent the even.

Although a variety of other techniques are also in use, these methods predominate.

IT'S ALL RELATIVE

Bar-code decoding algorithms assume the velocity of the code scan and the size of the bar-space elements are unimportant. Provided the scanning velocity does not vary beyond a given parameter, the module relationship between bars and spaces can be determined by comparing the bar and space widths in the time domain.

Figure 2 presents the four most common bit-level encoding techniques. Some code structures, such as Two of Five (2/5), compare bar widths. A code such as Interleaved Two of Five (I 2/5) requires bar-to-bar and space-to-space comparison. Spaces are used in an identical fashion to the bars. Code 39 can be most easily understood as a series of dark and light bars rather than bars and spaces. Its wide and narrow elements, however, are interpreted the same as 2/5 and I 2/5.

Finally, codes such as UPC are structured so that a bar denotes a bit and a space denotes a zero. With UPC, width measurements are usually done by making comparisons between a bar and its associated space and the next bar and its associated space.

There are some characteristics that are important for evaluating bar-code symbologies. I'll introduce these concepts here and elaborate more fully on them later when I describe real bar-code symbologies.

First, some codes are classified as continuous and others are classified as discrete. These terms describe the way encoded characters are concatenated to form a multicharacter, bar-code label. In a continuous code, the intercharacter space is part of the code structure and must adhere to strict dimensional restrictions as defined in the code specification. In a discrete code, the intercharacter space is not part of the code and is allowed to vary within fairly wide dimensional limits.

Figure 3—*Two-of-Five code contains all the information in the width of the bars. The spaces function only as separators.*

Figure 4—*Interleaved Two-of-Five attains a higher density than Two-of-Five by using the spaces as well as the bars for encoding data.*

Secondly, some codes are designed in such a manner that they are self-checking. In other words, an algorithm can be applied to each character so that substitution errors can only occur if two or more elemental errors appear within a single character.

Finally, for added data security a check digit can be appended to the bar code. With some codes, this is mandatory, but is optional with others. In any case, detailed calculation methods are outlined in the respective bar-code specifications. It is important to realize that these check digits are treated as an inherent part of the bar code. If the check calculation does not yield the expected result, the reader does not decode the scan.

REAL BAR CODES

Two-of Five Code is one of the simplest and most straightforward bar codes. Having its origin in the late 1960s, this numeric-only code has been applied to a number of industrial applications and most recently has been used in sequentially numbering airline tickets.

Two-of-Five Code contains all the information in the width of the bars—the spaces function exclusively as separators. Two bar widths are defined as a wide bar typically three times the width of a narrow bar. Narrow bars are interpreted as zero and wide bars as one. The intervening spaces may be any width, but are typically equal to the narrow bars.

Data is encoded using a modified binary method in which the bar positions from left to right are assigned weighting factors of 1, 2, 4, 7, and parity. (Zero is an exception to this rule.) In addition to the numeric symbols, distinctive start and stop codes are defined for bidirectional scanning. The character set encoding for Two-of-Five Code is shown in Table 1.

From Table 1, you can see that there are two levels of algorithms for deciphering bar code. First, an algorithm is applied to recover the stream of one and zero bits. Second, these ones and zeros are combined to create the bar-code character set. Two-of-Five Code uses a simple modified binary approach in which conversion is direct. Other codes use different coding schemes or lookup tables.

Since the white spaces carry no information, it follows that their width within the characters is not critical. The spaces between characters can be loosely defined as well. Because of this, Two-of-Five Code is classified as discrete.

The Two-of-Five–Code structure is also self-checking since all characters are composed of two wide bars and three narrow bars. Notably, this is where the code gets its name—two of five bars must always be wide. A decoding error would require two independent printing defects within the same scan line, a condition which would dictate the alignment of a void on a wide bar with a spot on a narrow bar within the same character. A Two-of-Five bar code is shown in Figure 3.

Developed in 1972, Interleaved Two-of-Five Code attains higher character density than Two-of-Five Code by using the spaces as well as the bars for encoding data characters. The actual encoding methodology is identical to that used in Two-of-Five Code. This code has seen much use in warehousing, heavy industrial applications, and the automotive industry.

Bars are used for encoding those numbers that appear in the odd positions and the even-number positions are represented by spaces. As a result of this interleaving, the symbology requires that an even number of digits be encoded. If an odd number of digits must be represented, a leading zero initiates the data string.

Note that by placing information in the spaces as well as the bars, Interleaved Two-of-Five Code is no longer discrete. It does, however, retain the self-checking attributes of Two-of-Five Code. Figure 4 illustrates an Interleaved Two-of-Five bar code.

The full alphanumeric Code 39 (also known as 3-of-9 Code) was developed in 1975. Each stand-alone Code 39 character is represented by a group of five bars and four spaces. Two of these bars are wide and one of the four spaces is wide (i.e., there are three wide elements out of a total of nine elements).

Figure 5—*Each Code-39 character is represented by a group of five bars and four spaces. Two of the bars and one of the spaces are wide (giving three wide bars out of the nine elements, hence the name). Code 39 includes a full alphanumeric character set.*

This arrangement results in 10×4 possibilities, or 40 characters. Four extra characters ($, /, +, and %) are formed with all narrow bars and three wide spaces. The character set includes a single start/stop code (*) and 43 alphanumeric data characters. Code 39 is a discrete, self-checking code. Figure 5 shows a Code 39 bar code.

JUST AS PLANNED

Pick up a spec on any of the current bar-code symbologies and you will find everything—print-to-contrast ratio, dimensional information, reference-decoding algorithms—spelled out in great detail. Judging from the vast information, you might conclude that the evolution of the bar code had been carefully orchestrated and strictly regimented. The fact is, as in so many human endeavors, these specifications are an attempt to document what had taken place.

Technology often evolves in surprising and unexpected ways. Sometimes it takes on a life of its own. Few of us like to admit this, especially if we're charged with advancing the art. When we come up with something really good, the natural inclination is to say that it came out just the way we planned it.

Long before bar code became common in retail, it tracked the movement of millions of tons of freight. This 1959 implementation, developed by Sylvania, used a stationary reader. The reader included a Xenon light source and photomultipliers which sensed the reflected light from the bar code. The numeric symbology consisted of strips approximately ¾″ × 6″. These giant bar codes were scanned by moving past readers and were called the *Kartrak rail-tracking system*.

After overcoming numerous technical obstacles, the read rate was purportedly comparable with many of today's UPC scan rates. This ultimately led to the installation of about 1000 readers and the bar coding of approximately 1.8 million railroad cars. In addition, 200,000 piggyback tractor-trailer containers were also bar coded. The readers were usually situated at junctions, interchanges, and entrances to rail yards. Kartrak usage peaked in 1968 and then gradually declined due mostly to inadequate label maintenance. Today, only about 50 scanners are still in operation.

MULTIDIMENSIONAL CODES

The codes I've described so far use only one dimension for information storage. However, there is a limit to how much data can be encoded using this, essentially serial, method.

Using two dimensions for data storage results in a substantially higher volume of data-per-unit size. However, this does rule out the use of

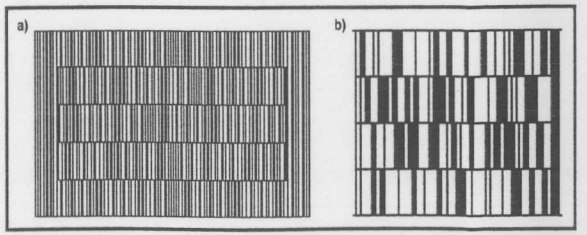

Figure 6—*Examples of two-dimensional codes include (a) CodaBlock, which is representing 65 characters here, and (b) Code 49, which is depicting 26 characters here.*

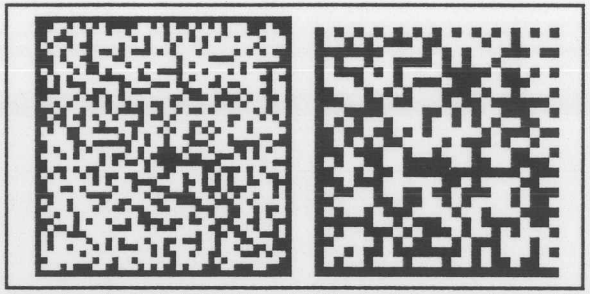

Figure 7—*Among other 2D codes, Veri Code (left) encodes 185 characters while its counterpart, Data matrix (right), represents 68 characters.*

an inexpensive, hand-held scanner. Moving-beam laser scanners or CCD scan heads prove suitable for this purpose. These devices are available in fixed-base or hand-held configurations.

The capability of packing more information into a given area can be useful in a number of ways. First, it simply stores more information. A less obvious method is to provide enhanced data integrity using the extra storage to contribute some data redundancy and, more importantly, error-detection and error-correction information. If the error-recovery mechanism is implemented properly, a label can be accurately read even if substantial portions have been damaged or obscured by dirt.

Developers of these 2D symbologies have taken different approaches to concatenating symbols as well as implementing data security and binary bit-encoding techniques. Figure 6 shows a number of 2D bar codes.

The simpler 2D codes start with existing symbologies and add the required control information to handle longer records and to ensure data accuracy. CodaBlock, which is based on Code 39 with additional control information and check digits, is representative of this approach.

Another strategy is to combine computer science disciplines with a formal data-redundancy and error-correction regimen that detaches the symbology from its traditional bar-code precursors. Only the optical scanning characteristics are retained. Figure 7 offers representatives of this class of inventive thinking. Judging by this strange landscape, all of a sudden it looks like we're a long way from the rail yards of 1959.

A SLIGHT REPRIEVE

Last time, I promised to put a microcontroller behind bars. Even though I tried to keep this month's introduction as brief as possible, I've managed to run out of space. Although I've covered a broad range of topics, I've done little to explain the technology in depth. As a result, our microcontroller has gained a slight reprieve.

To gain a better understanding of how a bar code really works, it would be instructive to pick one apart. Next month, I'll narrow my focus and concentrate on the hardware and firmware issues involved in putting a real microcontroller to work decoding real bar code.

John Dybowski is an engineer involved in the design and manufacture of embedded controllers and communications equipment with a special focus on portable and battery-operated instruments. He is also owner of Mid-Tech Computing Devices. John may be reached at (203) 684-2442 or at john.dybowski@circellar.com.

IRS

422 Very Useful
423 Moderately Useful
424 Not Useful

CONNECTIME

conducted by Ken Davidson

The Circuit Cellar BBS
300/1200/2400/9600/14.4k bps
24 hours/7 days a week
(203) 871-1988—Four incoming lines
Internet E-mail: sysop@circellar.com

With somwhat limited space this month, I decided to devote the whole column to just one message thread. RS-232, RS-422, and RS-485 connections are the most popular in use today for relatively slow asynchronous serial lines. However, not understanding how to ground systems that use them can cause major headaches. Check out some tips from the experts.

RS-232, -422, -485 grounding

Msg#:30409
From: IAN GARMAISE To: ALL USERS

This may seem like a silly question, but it is perplexing to me. I have a control system that uses RS-232-to-RS-422 converters. After a lot of on-line consultation, I went with the following wiring scheme:

I use two twisted pairs of #22 wire, overall shielded. The converter takes RJ-11 connectors. The vendor supplied RJ-11-to-DB-25 female connectors for the RS-232 side. I made RJ-11-to-DE-9 connectors to connect the adapter to the RS-422 cable. I used shielded DE-9 connectors. On one side, I connected the ground directly from the converter board to water-pipe ground. I did not connect ground on this side to the cable shield. On the other end, I connected the ground from the converter board to the PC's metal case, and I also connected the shield wire from the cable to the same case.

What's the right way to deal with the shield wire when you're attaching a metal DE-9 cover? I just wrapped it back, attached it to the cover with one screw, and ran a wire from the other screw to my case ground. Somehow this doesn't look right. I couldn't solder the wire to the cover cause I couldn't get the cover hot enough. Any suggestions?

By the way, the RS-422 works fine as long as I use surge protectors with my cheap 9-V power supplies on each end, and as long as both sides are grounded. Without a ground on both sides, the RS-422 didn't work, which makes sense.

Msg#:30908
From: JAMES MEYER To: IAN GARMAISE

Not to me, it doesn't.

The way I understand it, there should be no need for a "ground" in a connection that is truly RS-422 based.

I would be "very" careful about "water-pipe grounds" if I were trying to implement an RS-422 connection.

The whole point of RS-422 is its balanced, push-pull, differential nature and the fact that it doesn't need any "grounds." If you need a "ground" to make things work, then you must be doing something wrong.

Perhaps I missed something when I read your description of the connections you made. Please expand a little on "exactly" what you've got in the way of converters and how they're hooked up.

Msg#:31048
From: IAN GARMAISE To: JAMES MEYER

I basically was following some precise instructions from a person on the CompuServe Eng forum. He said that the differential voltages used in RS-422 still need a reference ground. His instructions were as follows:

1. If concerned about ground loop, use isolated RS-422 drivers and receivers, and CONNECT the (isolated) signal ground of each driver and receiver to provide a reference ground (the RS-422 doc does in fact show a connected signal ground.

2. If ground loop is not an issue, then connect the ground of each driver and receiver to mains ground at each end (this is what I did). As for the shield ground, lots of difference of opinion here, but the consensus of the noise books I borrowed and my converter supplier is to ground the shield at one end to the mains ground.

The CompuServe person said that sometimes RS-422 drivers connected without ground reference will appear to work correctly, but will fail eventually.

On one end, I have a PC serial RS-232 port. On the other end, I have a mag card reader with an RS-232 interface. They are connected as I described above.

On both ends, I'm using Tripp Lite surge protector/noise filters with the 12-VAC/9-VDC power supplies to the converters. While testing I discovered that as the CompuServe person predicted, transmission was impossible if I disconnected the signal ground at one end.

Msg#:31960
From: JAMES MEYER To: IAN GARMAISE

Sorry. Your CompuServe person is entirely right, and I was entirely wrong.

No. I can't leave it at that....

The reason I put "ground" in quotes above, is because

Micros Behind Bars

EMBEDDED TECHNIQUES

John Dybowski

To gain a better understanding of how bar code works, John picks one apart. He then looks at the hardware and firmware required to get a real microcontrollor decoding real bar code.

In last month's column, I looked at a number of media that are commonly employed in the field of Auto ID with a special emphasis on bar code. I touched on everything from the giant bar codes on rail cars, which move past xenon scanners, to two-dimensional wonders, which look more like artwork than encoded information. The range of complexity spanning the various symbologies collectively called bar codes is quite expansive.

And, as I pointed out last time, that range of complexity hinges on the fact that the industry is centered primarily around economic rather than technological concerns. Because of this practical focus, many of the older, simpler symbologies are still used heavily to this day. Codes such as Interleaved Two of Five exist side by side with such fiendishly complex multidimensional representations as VeriCode. Simply put, these older codes are kept around because they still serve their purpose well.

With the emphasis on technology being especially strong in the computer field, it's too easy to forget what pays the bills. Auto ID represents many technical fields pressed to serve the industrial and financial sectors. The bottom line is results, and many of these applications do just fine with a moderate dose of technology.

To those technology zealots who question how processors like the 8051 and 6805 not only survive but prosper, the answer is simple. They reliably provide useful services at low cost. In fact, 8051-class processors offer more performance than is needed for many applications. Bar-code readers are an example of this type of commodity.

CODE 39

Code 39 is a bar-code symbology with a full alphanumeric character set. A unique start/stop code (*) and seven special characters (– . $ / + % and space) are also included in the character set. The name *39* is derived from its code structure of three wide elements out of a total of nine. These nine elements are composed of five bars and four spaces.

Char.	Pattern	Bars	Spaces	Char.	Pattern	Bars	Spaces
1		10001	0100	M		11000	0001
2		01001	0100	N		00101	0001
3		11000	0100	O		10100	0001
4		00101	0100	P		01100	0001
5		10100	0100	Q		00011	0001
6		01100	0100	R		10010	0001
7		00011	0100	S		01010	0001
8		10010	0100	T		00110	0001
9		01010	0100	U		10001	1000
0		00110	0100	V		01001	1000
A		10001	0010	W		11000	1000
B		01001	0010	X		00101	1000
C		11000	0010	Y		10100	1000
D		00101	0010	Z		01100	1000
E		10100	0010	-		00011	1000
F		01100	0010	.		10010	1000
G		00011	0010	SPACE		01010	1000
H		10010	0010	*		00110	1000
I		01010	0010	$		00000	1110
J		00110	0010	/		00000	1101
K		10001	0001	+		00000	1011
L		01001	0001	%		00000	0111

Table 1—*The Code 39 encodable character set consists of 10 numeric digits, 26 alphabetic characters, and 8 special characters.*

Unlike some of the other more awkward bar codes, Code 39 uses only two element widths. These are usually simply described as narrow and wide. Using the normal convention, a narrow bar or narrow space is called the *x dimension*. All *x* dimensions must be of equal size within the symbol. The dimension of wide bars and spaces is a multiple of *x*. This ratio can vary within certain proportional limits but, once selected, must remain consistent throughout the symbol. Generally, a wide-to-narrow ratio in the range of 2:1 to 3:1 is acceptable for most Code-39 symbols.

The combination of narrow and wide elements in a Code-39 character always consists of six narrow and three wide elements. A space is included between characters as a separator. No information is contained in the space; it functions only to delimit the characters from each other.

A special code (an ASCII *) is defined as a start/stop character. The purpose of this code is to identify the leading and trailing ends of a bar-code symbol. The bar-space pattern of this code is unique and allows the symbol to be bidirectionally scanned.

Table 1 shows the Code-39 character assignments for all available codes. Note how the last four codes in the table "don't fit" the established coding pattern. Interestingly, if you take away these nonconforming characters you end up with 39 characters. Rumor has it that these 39 characters composed the original character set and are the basis for the Code-39 name. Whatever the case, Figure 1 offers an example of how to decode a Code-39 character "A".

Code 39 is classified as a discrete code since each encoded character is capable of standing alone. That is, the intercharacter space (or gap) is not considered an integral part of the character code and, as a result, enjoys somewhat loose tolerances. The minimum intercharacter width is the *x* dimension and the maximum is 3*x*.

Combining the desirable discrete attribute with a fixed structure (3 wide elements out of 9) results in code that is classified as self-checking. With this feature, the possibility of a missed decode is much less likely since a substitution error can only occur if two or more elements are misinterpreted. This could happen, for example, if a spot on a narrow bar lined up with a void on a wide bar and the resulting pattern turned out to be a legal-character depiction.

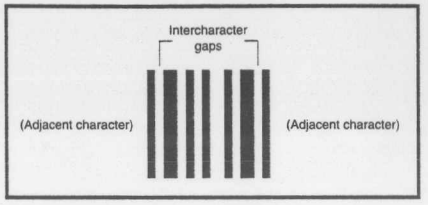

Figure 1—*Each Code-39 character is represented by five bars and four intervening spaces. This symbol represets the character "A".*

Another benefit of discrete codes is that they are well matched to certain printing processes. Some types of printers can maintain very tight resolution between elements within a character but are unable to maintain such accuracy in the space between characters. Obviously, these printers are fixed-font devices in which each character code is fully formed. This ensures that tolerances are held tightly within each character. The space between characters is dependent on the printer's mechanical motion and therefore less precise.

In addition to the bar-space pattern that makes up a bar-code character and the intercharacter gaps that delimit these characters, there is one more component to a bar-code label. Bar code must be framed with areas free of any printing on either side of the "picket fence" pattern. This region is referred to as the *quiet zone*.

Now, with this information we can take the pattern of ones and zeros to assemble a start code, some data characters, and a stop code. Framing this with the requisite quiet zones results in a standard bar-code label. These elements are depicted in Figure 2.

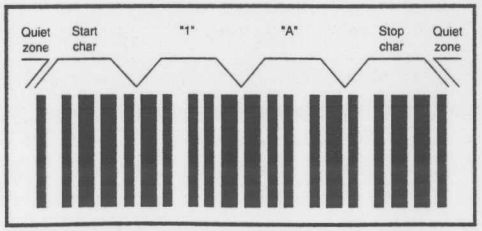

Figure 2—*Quiet zones, start/stop codes, and data codes constitute the elements of a bar code. The encoded information here is "1A".*

HAND SCANNING

Many methods exist for converting a bar code's optical information to an electrical form suitable for input into a computer. In all cases, the printed pattern of bars and spaces is converted into a binary bitstream as it is scanned physically or by purely electrical means. Since this data is transformed into the time domain, the bar-code processor must proceed by first recording timing information relative to each bar or space event.

Although some autoscanning readers are very accurate in their initial and absolute scan velocities, this is not a requirement. The main feature these devices offer is their rapid repetitive scanning action. Combined with a slight dither of the light source, the same symbol can be scanned numerous times through slightly different paths until a good read is recorded. This multiple scanning illustrates the data redundancy that is built into the vertical dimension of a bar code.

This redundant data can be used with a hand-held scanner as well. In the event of a decode failure, the natural inclination is to scan the label again. In this case, it is highly unlikely that the same part of the label will be scanned a second time.

Some applications require the use of noncontact automatic scanners.

The two-dimensional bar codes I described last month certainly demand this caliber of performance. More conventional bar codes may also dictate the use of such sophisticated devices as well. For instance, more complex devices must be used for tracking materials on rapidly moving conveyer belts, high-volume, point-of-sale operations, and long-range, point-and-shoot warehouse applications. Since this is a field in which high and low tech coexist side by side, dealing with conventional bar code in unique situations is possible.

Low-tech devices usually refer to hand-held bar-code wands. Of course, this distinction is purely relative and does not imply that such devices suffer from a lack of technological elegance. The fact is, until recently, coercing a clean stream of bits from a bar-code front end required a significant effort using optics and analog electronics.

The vagaries of these disciplines have been brought in check as is evident in modern, digital-output bar-code wands. Bar-code wands operate directly from a 5-V logic power supply and output a digital representation of the symbol being scanned. To facilitate an interface to a variety of different decoders, the output stage often uses an open-collector driver.

There are a number of parameters that must be considered when specifying the optical characteristics of a bar-code wand. Luckily, industry standardization has limited the number of permutations. Briefly, the optical wavelength can be centered in the visible (red) or infrared spectrum. The advantage of using visible light is that if the bar-code label looks fine to you, it should appear likewise to the wand.

The other thing you must be concerned with is the optical aperture size. A small spot size responds accurately to bar edges, but also picks up small spots and voids. Conversely, if the spot size is larger than the smallest bars and spaces, then the wand will have difficulty resolving the pattern. An aperture size about 0.8x works well for most codes. Here again, standardization limits the choices between high resolution (6 mil) and low resolution (10 mil).

Listing 1—*The five basic steps involved in decoding Code-39 can be implemented in C.*

```
#pragma large code

/* Constants */
#define StartCode '*'  /* Code 39 start code */
#define StopCode '*'   /* Code 39 stop code */
#define NoSample 0 /* Sample count end mark */
#define NoCode 0   /* No-translate return code */
#define NoDecode 0 /* No-decode return code */

/* Global variables */
unsigned int SampleData[512];          /* Raw sample count buff */
unsigned int *SamplePtr;               /* Pntr into sample buff */
unsigned int SampleCount;              /* Number of samples */
unsigned char DecodeData[33];          /* ASCII decode buffer */

/* External references */
extern unsigned char Decode39(void);   /* Main decode routine */
extern unsigned char DecodeChar(void); /* Bar to ASCII decode*/
extern void ReverseSamples(void);      /* Sample buff reversal */

/* Code 39 decode routine */
unsigned char Decode39(void)
{
    unsigned char DecodeCount, DecodeByte;

    SampleCount = 0;
    while (SampleData[SampleCount] != NoSample)
        SampleCount++;

    if (SampleCount < 27)
        return NoDecode;               /* Not enough samples */

    SamplePtr = &SampleData[0];
    if (DecodeChar() != StartCode) {   /* Check start code */
        ReverseSamples();              /* Try reverse direction */
        SamplePtr = &SampleData[0];
        if (DecodeChar() != StartCode)
            return NoDecode;           /* Can't find start code */
    }

/* Main decode loop */
    DecodeCount = 0;
    while ((*SamplePtr++ != NoSample) && ((DecodeByte =
            DecodeChar()) != NoCode)) {
        if (DecodeByte != StopCode)      /* Store data character */
            DecodeData[DecodeCount++] = DecodeByte;
        else {
            DecodeData [DecodeCount] = 0;
            return DecodeCount-1;        /* Stop code found */
        }
    }
    return NoDecode;   /* Unable to decode */
}

/* Generate ASCII character from bar/space pattern */
unsigned char DecodeChar(void)
{
    static code unsigned char BarTable[4][25] = {
        {0,0,0,'7',0,'4','0',0,0,'2','9',0,'6',
                0,0,0,0,'1','8',0,'5',0,0,0,'3'},
        {0,0,0,'G',0,'D','J',0,0,'B','I',0,'F',
                0,0,0,0,'A','H',0,'E',0,0,0,'C'},
        {0,0,0,'O',0,'N','T',0,0,'L','S',0,'P',
                0,0,0,0,'K','R',0,'O',0,0,0,'M'},
        {0,0,0,'-',0,'X','*',0,0,'V',' ',0,'Z',
                0,0,0,0,'V','.',0,'Y',0,0,0,'W'}
    };
```

(continued)

Listing 1—*continued*

```
    unsigned int *TempPtr, Threshold;
    unsigned char Bars, Spaces, c;

/* Generate reference threshold */
    TempPtr = SamplePtr;
    Threshold = 0;
    for (c = 0; c < 9; c++) {
        if ((*TempPtr) == NoSample)
            return NoCode;
        Threshold += *TempPtr++;
    }
    Threshold /= 8;
    Bars = 0;
    Spaces = 0;

/* Build binary bar/space image */
    for (c = 0; c < 4; c++) {
        if (*SamplePtr++ > Threshold)
            Bars |= 1;
        Bars <<= 1;

        if (*SamplePtr++ > Threshold)
            Spaces |= 1;
        Spaces <<= 1;
    }

    if (*SamplePtr++ > Threshold)
        Bars |= 1;
    Spaces >>= 1;
```

(continued)

SAMPLING

The first step to decoding a bar code is acquiring the bar-space data. More specifically, information describing the bar-space widths must be recorded. This sampling can be performed in a number of different ways and, as usual, the appropriate method depends on what else is expected of the system.

Dedicated implementations, in which the system can dedicate all processor resources to sampling, permit the use of a simple software loop for counting the bar-space durations. Alternatively, it may be desirable to give the processor assistance from a hardware timer in lieu of using a software-based timing loop. Both these cases rely on the premise that the system can somehow vector off to the sample loop before too much of the first bar is lost.

If it is possible that the system may be off performing other tasks when the bar-space data starts coming in, then obviously the processor must suspend these operations promptly or

the first sample will be hopelessly distorted. If this is the case, you can use an interrupt to simply yank the processor into a dedicated sample loop where it stays until the sampling phase completes.

If you've got to stay live while servicing other real-time events, then there's no choice but to sample completely under interrupt control. This technique mandates the use of a hardware timer that is stopped, read, and rearmed every time an interrupt event occurs. You must provide a means of generating an interrupt on each transition, and the interrupt should be given high priority. Also, most timer systems have the capability of interrupting on terminal count. This is exactly what you want to pull you out of sampling after you've entered the trailing quiet zone and data transitions have ceased.

Some systems may have to deal with real-time events that are more critical in nature than the incoming bar-code data. This situation can be handled provided your processor has a timer-capture system. In such a system, the sample count is copied into a capture register from a free-running timer without stopping the timer. This happens automatically under control of a hardware pin that can also be used to assert an interrupt when a transition event occurs. The processor has until the next event to read the captured count before it is overwritten, resulting in a sample loss.

Very accurate timing measurements can be achieved using such a system. Of course, the sample buffer requires some manipulation to adjust all samples to look like zero-referenced up counts. Also, setting the proper duration for the timer-overflow interrupt requires additional overhead. (For thoughts on general-purpose sampling techniques, take a look at my column in *INK* 30.)

For my sampling routine, I'm taking advantage of the simplicity of the dedicated software method, although you'd seldom be able to use such a primitive technique in a real-world application. Since I'm primarily interested in showing you how to decode bar code, I won't waste space going into bar-code sampling in any detail. For information purposes, let me briefly describe the steps taken by my rudimentary software sample loop.

Coming from an idle state, control is transferred to the sample routine on detection of a data transition (the first bar). The routine now initializes some general variables and starts incrementing a counter register until the data line changes to the opposite polarity. Once this change occurs, the count is stored, the storage pointer incremented, and the procedure begins all over again. This cycle continues until the counter reaches some terminal value (due to lack of transitions) at which point the trailing quiet zone is recognized and the routine terminates.

Since the count interval is referenced to the loop time, this parameter can be tuned to accommodate the range of values which are encountered. Assume a nominal x dimension of 0.020", a wide-to-narrow ration of 3:1, and a 10x quiet zone. A realistic scan rate would typically fall in the range of 5–30" per second.

Listing 1—*continued*

```
/* Now do lookup based on bar-space combination */
    if (Bars > 24)
        return NoCode;

    switch (Spaces) {
        case 0x4:   return BarTable[0][Bars];  /* 0100b */
        case 0x2:   return BarTable[1][Bars];  /* 0010b */
        case 0x1:   return BarTable[2][Bars];  /* 0001b */
        case 0x8:   return BarTable[3][Bars];  /* 1000b */
        case 0xe:   {                          /* 1110b */
                    if (Bars == 0)
                        return '$';
                    break;
                    }
        case 0xd:   {                          /* 1101b */
                    if (Bars == 0)
                        return '/';
                    break;
                    }
        case 0xb:   {                          /* 1011b */
                    if (Bars == 0)
                        return '+';
                    break;
                    }
        case 0x7:   {                          /* 0111b */
                    if (Bars == 0)
                        return '%';
                    break;
                    }
        default:    return NoCode;
    }
}

/* Do sample buffer reversal */
void ReverseSamples(void)
{
    unsigned int *Ptr1, *Ptr2, Count, Temp;

    Count = SampleCount-1;
    Ptr1 = &SampleData[0];
    Ptr2 = &SampleData[Count];

    for (Count /= 2; Count != 0; Count--) {
        Temp = *Ptr2;
        *Ptr2-- = *Ptr1;
        *Ptr1++ = Temp;
    }
}
```

To accommodate these parameters, the sample counter is 16 bits wide. The sample loop time is set to about 2.5 µs. Terminal count is reached after an interval of 10 ms, and in the absence of transitions, this is the overflow count. To save space for the decoding algorithm, the sample routine listing is not presented here. However, the BAR.ZIP archive is available on the Circuit Cellar BBS and contains this and related modules. (If you do decide to examine the sampling routine, remember that it is set up to run on a 25-MHz DS80C320.)

DECODING 39

In keeping with my goal of providing a simplified firmware presentation, I will demonstrate the essence of Code 39's decoding algorithm. This is in fact an implementation of the logic described in the *Automatic Identification Manufacturers (AIM) Reference Decode Algorithm* for USS-39.

At this point, it would be useful to make a couple of general observations. This decode algorithm presents the basic steps for deciphering a Code-39 symbol. The underlying logic is sound, but incomplete. As the AIM specification points out, you would undoubtedly want to add secondary checks for acceleration, intercharacter gap, and absolute dimensions for any serious application. You should also realize that these secondary checks and balances can generate as much code as the algorithm. As a result, the logic of the algorithm can become totally obscured.

The other relevant issue falls smack in the realm of advancing the state of the art. It's not unusual to encounter bar-code labels that don't meet specifications. This may be due to dimensional-tolerance problems, poor print-contrast ratio, inadequate quiet zone, and suchlike. If some clever programmer comes up with a superior algorithm which consistently reads deficient labels (one that doesn't result in an increase of missed decodes, of course), this has an unsettling effect on the status quo. These things happen and illustrate the fact that meeting the specification should be just a starting point.

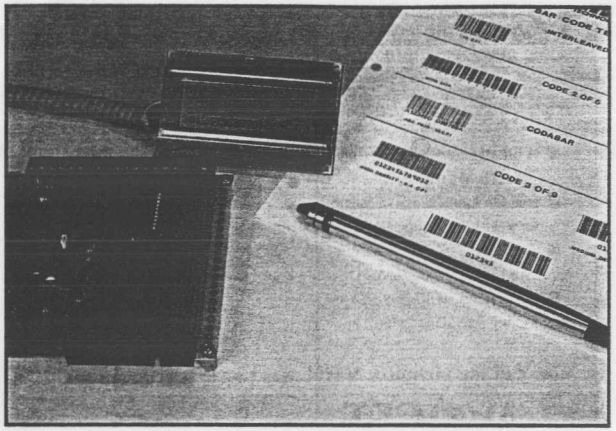

Photo 1—*Running the test code on a DS80C320 processor yields properly decoded data displayed on the LCD display.*

The basic steps in decoding Code 39 are:

1) Confirm a leading quiet zone
2) For each character,
   - measure and assign total character width to *S*
   - compute threshold, $T = \frac{5}{8}$
   - build binary bit strings for bars and spaces
   - determine if the pattern matches a valid character
3) If the first character is not a start/stop code, reverse buffer and try again
4) Read until valid start/stop code is found (or until out of samples)
5) Perform secondary checks

These basic steps are implemented in the source code contained in Listing 1. This C implementation begins with the main decode function called `Decode39`. This function first counts the number of samples and determines if there are enough to continue. If the minimum number of samples is available, the `DecodeChar` function is invoked. This function actually does the work.

`DecodeChar` begins by summing the nine samples that (presumably) compose a character. A constant is applied to this sum resulting in the narrow- or wide-reference threshold. The code sequentially compares the character's sample counts to this threshold and builds a binary representation of the bars and spaces. Using the binary-space pattern, a switch statement is performed. The first four cases handle the "normal" Code-39 characters and isolate ASCII code to the look-up table.

The table is in the form of a two-dimensional array that consists of 4 arrays of 25 elements each. Illegal codes are denoted by null codes. The four remaining "special" space patterns are directly validated and translated in the switch. The function now terminates and returns either a decoded ASCII code or an error code to the caller.

At this point in `Decode39`, the only valid character is a start code. If anything other than a start code is returned, the function assumes that this may have been a reverse scan and inverts the sample buffer. Following this, the pattern-matching procedure is performed again. If a start code is not recognized this time, the function terminates and indicates a no-decode to the caller.

If a start code is found, then the code falls through and indexes past the intercharacter gap and invokes `DecodeChar` again. If a displayable code is returned, it is placed into the `DecodeData` array. An invalid code causes the function to terminate immediately and return indicating a no-decode. Detection of a stop code marks the completion of a good decode sequence. In this case, a trailing null is appended to the decoded data, and a value indicating the number of characters is returned to the caller.

DISCLAIM THIS

The functions I've presented all work individually and together. As evidence, Photo 1 shows the ec.32 SBC serving as the test bed in developing and testing the demonstration algorithms. The apparent performance of the system is actually quite good, and I encountered no problems with the system once I got the basic functions operational.

Where my discomfort lies is in the routines. I am well aware of the code's limitations, deficiencies, and omissions. That's not to say that I don't have a solid foundation on which to build, but clearly, the code is not finished.

From the user's perspective, this is not at all evident. At times like this, I wonder what lurks under the hood of some of the commercial software and systems. At least when I give you a weak algorithm, you get a disclaimer up front.

John Dybowski is an engineer involved in the design and manufacture of embedded controllers and communications equipment with a special focus on portable and battery-operated instruments. He is also owner of Mid-Tech Computing Devices. John may be reached at (203) 684-2442 or at john.dybowski@circellar.com.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

I R S

434 Very Useful
435 Moderately Useful
436 Not Useful

CONNECTIME

conducted by Ken Davidson

The Circuit Cellar BBS
300/1200/2400/9600/14.4k bps
24 hours/7 days a week
(203) 871-1988—Four incoming lines
Internet E-mail: sysop@circellar.com

With the start of our quarterly home automation inserts in this issue, I thought it only appropriate to spend this month's column dealing with home automation threads from the BBS. In the first discussion, we take a look at some of the potential pitfalls in trying to add intelligence to an HVAC system. While hot-water baseboard setups aren't particularly difficult to deal with, forced-air systems can be quite tricky.

In the other thread, we tackle a problem that comes up all the time in every on-line home automation forum I follow: flaky X-10 behavior. There is nothing cut and dry about power-line communications.

Fan control and HCS II

Msg#: 9252
From: DAVID WURMFELD To: KEN DAVIDSON

Is there a fan controller interface to the HSC II? I want to control the speed of my forced (hot/cold) air system. I am also looking for (digitally?) controlled air duct flapper valves. Eventually I would like to "shut down" the A/C in some rooms and not in others, so I would have to slow down the one service fan so as to not overpressure the reduced system. Any ideas for the "analog challenged"?

Msg#:11727
From: BILL NEUKRANZ To: DAVID WURMFELD

I'd be careful about trying to control furnace fan speed. Both your A/C and furnace units require good airflow to operate within safe limits.

You're correct to be concerned about pressure build up when running a zoned HVAC system. You should also be concerned about the liquid freon line getting too cold, eventually causing A/C compressor failure. And, during the heating season, you should be concerned about the furnace heat exchanger getting too hot. I'm operating a five-zone system for a 3400-square-foot home, with a single HVAC unit, and have protection for all three of these situations.

For the pressure, you can simply always run a "dump zone." That is, a zone that's always open in addition to any other zone. In my five-zone system, that would mean the minimum number of zones open would be two. Another technique is to install a bypass duct that starts at the same point as all of your other ducts, and ends at the intake side of the furnace blower. In the middle of this duct you install a pressure valve. Adjust the valve such that it's closed when all zones are calling for air. I'm using the bypass valve solution.

The bypass duct also helps protect the A/C compressor by increasing air flow across the evaporator coil, keeping the liquid freon line from getting too cold. Additionally, I mounted a simple 45° temperature sensor switch directly onto the freon line. The switch is interfaced into the zoning controller. When the switch opens up at 45°, all air duct flapper valves open, maximizing air flow.

For winter heating, I've had to use a dump zone in past years. Otherwise, the furnace emergency-high-heat cutout switch would operate. I wasn't too interested in essentially "modulating" the furnace using this emergency protection. You'd have the same problem if you reduced fan speed without correspondingly reducing burner operation. This year, I have just finished installing a temperature sensor into the plenum distribution area that supplies all of the duct work. Once I figure out what is a safe temperature level, I'll program my controller to open up more zones when it gets too hot.

Some other things I'm doing that may give you some ideas for HVAC zoning:

1. I'm using balloons, not mechanical dampers, for what you're calling "flapper valves." They're very easy to install, especially for retrofit situations (like if your house is already built). I use an air pump to inflate or vacuum them. Much less expensive that the mechanical dampers. I got the equipment from Enerzone Co., in Dallas. All U.L. listed, too.

2. Get yourself a good controller if you're going to have three or more zones. I'm using an Enerstat five-zone controller. Works with heat pumps and forced air. Handles multiple stages of cooling (our A/C unit is a two speed unit). Also has digital inputs for unoccupied, high temperature limit, low temperature limit, and smoke alarm. I have all of these inputs connected to my home controller (not an HCS II, but performs similarly). The controller will make sure you don't overcycle the compressor, always have at least one zone open, shuts down and opens balloons in case